

# FASTER ARITHMETIC FOR NUMBER-THEORETIC TRANSFORMS

DAVID HARVEY

**ABSTRACT.** We show how to improve the efficiency of the computation of fast Fourier transforms over  $\mathbf{F}_p$  where  $p$  is a word-sized prime. Our main technique is optimisation of the basic arithmetic, in effect decreasing the total number of reductions modulo  $p$ . We give performance results showing a significant improvement over Shoup's NTL library.

## 1. INTRODUCTION

An important problem in computational number theory and cryptography is the efficient implementation of modular arithmetic. A typical implementation strategy is to represent elements of  $\mathbf{Z}/N\mathbf{Z}$  by residues in a standard interval, such as  $[0, N)$  or  $[-N/2, N/2)$ , and to reduce intermediate results to this interval after each operation in  $\mathbf{Z}/N\mathbf{Z}$ , such as addition or multiplication.

In many algorithms one can obtain a substantial constant factor speedup by delaying the modular reduction until after several arithmetic operations have been performed, by taking into account the bit-size of intermediate results. For example, to compute a dot product  $\sum_i a_i b_i$ , a fundamental operation in linear algebra, one may accumulate terms in batches, using ordinary integer (or even floating-point) arithmetic, and perform the reduction modulo  $N$  after each batch [1].

The aim of this paper is to describe a strategy for reducing the number of modular reductions in the computation of a discrete Fourier transform over a finite field, also known as a *number-theoretic transform* (NTT). The NTT has a vast range of applications; we mention here only fast multiplication of large integers or polynomials [5].

For simplicity we restrict to the following situation. Let  $\beta$  describe the machine word size, for example  $\beta = 2^{32}$  or  $\beta = 2^{64}$ . We work over  $\mathbf{F}_p = \mathbf{Z}/p\mathbf{Z}$  where  $p$  is a word-sized prime, and we assume that  $p \equiv 1 \pmod{L}$ , where  $L = 2^\ell$  is the transform length. This ensures that  $\mathbf{F}_p$  contains a primitive  $L$ -th root of unity; we fix one of these, denoted  $\omega$ . The NTT is the map  $(\mathbf{F}_p)^L \rightarrow (\mathbf{F}_p)^L$  defined by

$$b_j = \sum_{i=0}^{L-1} \omega^{ij} a_i, \quad 0 \leq j < L.$$

Equivalently, this is the map that evaluates the polynomial  $a_0 + a_1x + \dots + a_{L-1}x^{L-1}$  at the points  $1, \omega, \omega^2, \dots, \omega^{L-1}$ .

It is well known that the fast Fourier transform (FFT) can be used to compute the NTT using  $O(L \log L)$  operations in  $\mathbf{F}_p$ . For completeness, a simple in-place iterative radix-2 FFT algorithm is shown in Algorithm 1.

---

**Algorithm 1:** Simple FFT

---

**Input:**  $\omega \in \mathbf{F}_p$  and  $x = (x_0, \dots, x_{L-1}) \in (\mathbf{F}_p)^L$   
**Output:** DFT of  $x$  with respect to  $\omega$ , in bit-reversed order

```

1 for  $i \leftarrow 1, 2, \dots, \ell$  do
2    $\zeta \leftarrow \omega^{2^{i-1}}$ 
3    $m \leftarrow 2^{\ell-i-1}$ 
4   for  $0 \leq j < 2^i$  do
5      $t \leftarrow 2jm$ 
6     for  $0 \leq k < m$  do
7        $\begin{bmatrix} x_{t+k} \\ x_{t+k+m} \end{bmatrix} \leftarrow \begin{bmatrix} x_{t+k} + x_{t+k+m} \\ \zeta^k(x_{t+k} - x_{t+k+m}) \end{bmatrix}$ 
8     end
9   end
10 end
```

---

Our main focus in this paper is on the *butterfly operation* in line 7, which computes  $(X, Y) \mapsto (X + Y, W(X - Y))$  for some fixed root of unity  $W \in \mathbf{F}_p$  (a suitable power of  $\omega$ ). At first glance this requires one modular addition, one modular subtraction, and one modular multiplication, and this is how the butterfly is usually implemented. Of course these modular operations are themselves implemented on modern microprocessors using more basic primitives. For example, a modular addition is usually implemented as an ordinary integer addition, followed by a comparison with  $p$ , followed by a conditional subtraction. In this paper we investigate the butterfly at this lower level, showing how to streamline the implementation to reduce the number of comparisons and conditional operations. Another interpretation is that we have reduced the number of modular reductions.

## 2. A TYPICAL BUTTERFLY IMPLEMENTATION

Victor Shoup's NTL (Number Theory Library) [4] is a popular C++ library used in computational number theory. It makes heavy use of the NTT. Its implementation of the butterfly  $(X, Y) \mapsto (X + Y, W(X - Y))$  may be expressed by the pseudocode shown in Algorithm 2. It has been simplified to focus attention on the arithmetic aspects, ignoring issues like loop unrolling, software pipelining, and locality. All variables represent register-sized quantities.

---

**Algorithm 2:** NTL butterfly implementation

---

**Constants:**  $p < \beta/2$ 

$$0 < W < p$$

$$W' = \lfloor W\beta/p \rfloor, 0 < W' < \beta$$

**Input:**  $0 \leq X < p$ 

$$0 \leq Y < p$$

**Output:**  $X' = X + Y \pmod{p}$ 

$$Y' = W(X - Y) \pmod{p}$$

```

1  $X' \leftarrow X + Y$ 
2 if  $X' \geq p$  then  $X' \leftarrow X' - p$ 
3  $T \leftarrow X - Y$ 
4 if  $T < 0$  then  $T \leftarrow T + p$ 
5  $Q \leftarrow \lfloor W'T/\beta \rfloor$ 
6  $Y' \leftarrow (WT - Qp) \bmod \beta$ 
7 if  $Y' \geq p$  then  $Y' \leftarrow Y' - p$ 
8 return  $X', Y'$ 

```

---

Lines 1–2 compute the sum  $X + Y$  and reduce it modulo  $p$  to the standard interval  $[0, p)$ , using the assumption  $p < \beta/2$  to avoid overflow in the first line. Lines 3–4 compute  $T = X - Y \pmod{p}$  in the same way, assuming that  $T$  has a signed type for the comparison.

Lines 5–7 compute the product  $WT \pmod{p}$ . Line 5 first generates an estimated quotient  $Q$ . The expression  $\lfloor W'T/\beta \rfloor$  represents the high word of the product  $W'T$ . We assume that this high word is computed efficiently using a hardware multiply instruction. By the definition of  $W'$  and  $Q$  we have

$$0 \leq \frac{W\beta}{p} - W' < 1, \quad 0 \leq \frac{W'T}{\beta} - Q < 1.$$

Multiplying by  $Tp/\beta$  and  $p$  respectively, and adding, yields

$$0 \leq WT - Qp < \frac{Tp}{\beta} + p < 2p < \beta.$$

In particular, line 6 correctly computes  $Y' = WT - Qp$ , and the single correction in line 7 suffices to reduce it into  $[0, p)$ .

The modular multiplication algorithm in lines 5–7 is attributed to Shoup in [2], but does not seem to have been published. It first appears in NTL version 5.4 in 2005. The use of a suitable precomputed approximation to  $W/p$  implies that only a single correction step (line 7) is necessary, and that the remainder is obtained using only multiplication modulo  $\beta$  (line 6), an advantage on processors that can compute the low word faster than the full product.

In our exposition we assumed for simplicity that  $p < \beta/2$ , but Niels Möller has pointed out (personal communication) that this can be improved. The modular subtraction can be made to work for any  $p < \beta$  by replacing the condition  $T < 0$  by  $X < Y$ , or indeed by using the borrow generated by the subtraction  $T = X - Y$ . The addition can be treated similarly by rewriting it as  $X' = X - (p - Y)$ . A more careful analysis of the candidate remainder  $WT - Qp$  then shows that the entire algorithm works for any  $p < \beta/\phi$  where  $\phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1.618$ .

### 3. A MODIFIED BUTTERFLY

In this section we propose several modifications to Algorithm 2. Our motivation is that the adjustment steps in lines 2, 4 and 7 are relatively expensive on modern microprocessors, compared to hardware integer multipliers, which in recent years have become very fast.

Our first observation is that Shoup's algorithm for computing  $WT \pmod{p}$  does not require that  $0 \leq T < p$ ; in fact the proof given above shows that it works for any  $0 \leq T < \beta$ . Therefore we may replace lines 3–4 by simply  $T \leftarrow X - Y + p$ , after which we have  $0 \leq T < 2p < \beta$ , and the algorithm proceeds as before. This simplification is not new, although it does not appear to be well known. It is not used in NTL. It was apparently used by Fabrice Bellard in a computation of  $\pi$  to 2.7 trillion decimal places in 2009 (personal communication).

Our second observation appears to be new, and is the main novelty introduced in the present paper. Namely, we may also remove the adjustment in line 7, provided that throughout the FFT we use a *redundant* representation for elements of  $\mathbf{F}_p$ . That is, instead of representing elements of  $\mathbf{F}_p$  by integers in  $[0, p)$ , we use the wider interval  $[0, 2p)$ , so each element has two possible representations. For this to work, we must modify the butterfly to accept *inputs* in  $[0, 2p)$ , and we must impose the stronger condition  $p < \beta/4$ . Pseudocode for the resulting butterfly is shown in Algorithm 3.

According to Jason Papadopolous (personal communication), around 1998 Ernst Mayer suggested the use of a redundant representation in the context of a fast Galois transform, i.e. a Fourier transform over  $\mathbf{F}_{q^2}$  where  $q = 2^{61} - 1$ . Our new algorithm may be regarded as a generalisation of this idea to the case of an NTT with arbitrary modulus  $p$ .

The proof of correctness is much the same as before. Lines 1–2 compute a representative for  $X + Y \pmod{p}$  in the interval  $[0, 2p)$ . Line 3 computes a representative  $T$  for  $X - Y \pmod{p}$  in the interval  $[0, 4p)$ ; this does not overflow as  $p < \beta/4$ . Lines 4–5 compute a representative for  $WT \pmod{p}$  in the interval  $[0, 2p)$ . Both outputs  $X'$ ,  $Y'$  lie in  $[0, 2p)$ , ready to be processed by a subsequent butterfly.

Depending on the needs of the NTT user, it may be necessary to normalise the final NTT output into the interval  $[0, p)$ , imposing an additional  $O(L)$  cost.

---

**Algorithm 3:** Modified Shoup butterfly

---

**Constants:**  $p < \beta/4$ 

$$0 < W < p$$

$$W' = \lfloor W\beta/p \rfloor, 0 < W' < \beta$$

**Input:**  $0 \leq X < 2p$ 

$$0 \leq Y < 2p$$

**Output:**  $X' = X + Y \pmod{p}, 0 \leq X' < 2p$ 

$$Y' = W(X - Y) \pmod{p}, 0 \leq Y' < 2p$$

```

1  $X' \leftarrow X + Y$ 
2 if  $X' \geq 2p$  then  $X' \leftarrow X' - 2p$ 
3  $T \leftarrow X - Y + 2p$ 
4  $Q \leftarrow \lfloor W'T/\beta \rfloor$ 
5  $Y' \leftarrow (WT - Qp) \bmod \beta$ 
6 return  $X', Y'$ 

```

---

## 4. VARIANTS

The same idea may be applied to butterflies using other modular multiplication algorithms. Algorithm 4 shows a variant using Montgomery multiplication [3].

---

**Algorithm 4:** Modified Montgomery butterfly

---

**Constants:**  $p$  odd,  $p < \beta/4$ 

$$0 < W < p$$

$$W' = \beta W \pmod{p}, 0 < W' < p$$

$$J = p^{-1} \pmod{\beta}$$

**Input:**  $0 \leq X < 2p$ 

$$0 \leq Y < 2p$$

**Output:**  $X' = X + Y \pmod{p}, 0 \leq X' < 2p$ 

$$Y' = W(X - Y) \pmod{p}, 0 \leq Y' < 2p$$

```

1  $X' \leftarrow X + Y$ 
2 if  $X' \geq 2p$  then  $X' \leftarrow X' - 2p$ 
3  $T \leftarrow X - Y + 2p$ 
4  $R_1\beta + R_0 \leftarrow W'T$ 
5  $Q \leftarrow R_0J \bmod \beta$ 
6  $H \leftarrow \lfloor Qp/\beta \rfloor$ 
7  $Y' \leftarrow R_1 - H + p$ 
8 return  $X', Y'$ 

```

---

Lines 1–3 are identical to the corresponding lines in Algorithm 3. Line 4 computes the product  $W'T$ , placing the low and high words of the result in  $R_0$  and  $R_1$  respectively, so that  $0 \leq R_1 < p$ . Line 5 computes  $Q = R_0/p \pmod{\beta}$  (this may be regarded as a 2-adic approximation to the quotient  $W'T/p$ ). Line 6 computes the high word of  $Qp$ , so that  $0 \leq H < p$ . We have  $Qp = R_0 \pmod{\beta}$ , so  $Qp = H\beta + R_0$ . Then  $W'T - Qp = \beta(R_1 - H)$ , and thus  $WT = R_1 - H \pmod{p}$ . This agrees modulo  $p$  with the value computed for  $Y'$  in line 7, which lies in the interval  $[0, 2p)$ . Usually in Montgomery's algorithm, a further comparison and conditional subtraction (or addition) is performed to normalise the result into  $[0, p)$ , but we have simply skipped that.

Compared to Algorithm 3, this butterfly algorithm has the advantage that only a single value  $W'$  must be stored in a table for each root of unity ( $W$  is not actually used in the algorithm). On the other hand, one of the multiplications modulo  $\beta$  has been replaced by a full product, which may be more expensive on some processors.

In some situations one requires a butterfly of the form  $(X, Y) \mapsto (X + WY, X - WY)$ . This appears if one switches from a 'decimation-in-frequency' transform to a 'decimation-in-time' transform. It also appears naturally in the inverse FFT obtained by running Algorithm 1 backwards. Algorithm 5 shows an analogue of Algorithm 3 for this case. The main difference is that elements of  $\mathbf{F}_p$  are represented by an integer in  $[0, 4p)$ , so each residue has four possible representatives.

---

**Algorithm 5:** Modified inverse butterfly

---

**Constants:**  $p < \beta/4$

$0 < W < p$

$W' = \lfloor W\beta/p \rfloor, 0 < W' < \beta$

**Input:**  $0 \leq X < 4p$

$0 \leq Y < 4p$

**Output:**  $X' = X + WY \pmod{p}, 0 \leq X' < 4p$

$Y' = X - WY \pmod{p}, 0 \leq Y' < 4p$

1 **if**  $X \geq 2p$  **then**  $X \leftarrow X - 2p$

2  $Q \leftarrow \lfloor W'Y/\beta \rfloor$

3  $T \leftarrow (WY - Qp) \pmod{\beta}$

4  $X' \leftarrow X + T$

5  $Y' \leftarrow X - T + 2p$

6 **return**  $X', Y'$

---

Line 1 reduces  $X$  into  $[0, 2p)$ , lines 2–3 compute  $T = WY \pmod{p}$  in  $[0, 2p)$ , and the remaining lines complete the calculation of  $X'$  and  $Y'$  in  $[0, 4p)$ . As in Algorithm 3, this strategy saves two modular reductions compared to the usual implementation.

## 5. IMPLEMENTATION AND PERFORMANCE

The practical benefit (if any) derived from the new algorithms depends heavily on the hardware used. To illustrate what may occur in practice, we performed some timing experiments on a 3.06GHz Intel Xeon (model X5675, ‘Westmere’ microarchitecture), a modern 64-bit processor.

We compared a C implementation of a number-theoretic transform incorporating Algorithm 3, a similar implementation of a transform based on Algorithm 2, and the FFT routine in NTL itself (version 5.5.2). The code is available from the author’s web site. It is moderately optimised: the inner loops are 2-way unrolled, and the last two layers of the FFT have dedicated loops. The values of  $W$  and  $W'$  are precomputed and stored in a table. Everything was compiled using GCC 4.6.3, with the `-O2` optimisation flag. Access to the high word of the product of two 64-bit integers is obtained using the compiler’s built-in support for 128-bit integer types.

We ran transforms of length  $2^{11} = 2048$ . This is long enough to avoid too much loop overhead, but short enough that all memory accesses hit the L1 cache, so we may ignore locality problems. For our code we used a 62-bit prime  $p$ , and for NTL we used a 50-bit prime (NTL supports moduli of only up to 50 bits on a 64-bit machine, for historical reasons related to floating-point arithmetic).

The NTL FFT routine also performs a bit-reversal of the input array and computes a table of roots of unity on each FFT invocation. To make a fair comparison with our code, which does not perform these steps, we removed their contribution in the timing data shown below.

TABLE 1. Cycles per butterfly for several implementations

NTL	12.0	(entire FFT)
Algorithm 2	10.1	(entire FFT)
	11.0	(inner loop)
Algorithm 3	6.9	(entire FFT)
	7.5	(inner loop)

For the lines labelled ‘entire FFT’, we measured the total time (CPU clock cycles) taken by the FFT, and divide by the number of butterflies performed, which is  $11 \times 2^{10}$ . Note that  $O(L)$  of the  $O(L \log L)$  butterflies have  $W = 1$ , and these are faster than the general butterfly with  $W \neq 1$ . For the lines labelled ‘inner loop’, we measured the speed of the main inner loop, which does not take advantage of any occurrences of  $W = 1$ .

Two features of the table are worth pointing out. First, our implementation of Algorithm 2 is competitive with NTL. In fact, it is slightly faster. We do not know the precise reason for this. The C compiler has many

choices in how it generates the machine code, and in some cases may generate suboptimal code. We may have simply been luckier than NTL in this case.

Second, Algorithm 3 outperforms Algorithm 2 by a factor of almost 1.5. In favourable circumstances, the Westmere processor can sustain a maximum throughput of one 64-bit multiplication every 2 cycles. If we assume that a butterfly requires three such multiplications, then the best we can expect is 6 cycles per butterfly. The 7.5 cycles reported above comes fairly close to this. We expect that a careful assembly implementation would get closer, and possibly even reach exactly 6, with all other operations executed in parallel with the multiplications. It seems unlikely that Algorithm 2 could achieve this speed.

#### ACKNOWLEDGEMENTS

The author thanks Tommy Färnqvist, Torbjörn Granlund, Niels Möller, Jason Papadopolous and Paul Zimmermann for stimulating conversations on this topic. The author was partially supported by the Australian Research Council, DECRA Grant DE120101293.

#### REFERENCES

1. Jean-Guillaume Dumas, Thierry Gautier, and Clément Pernet, *Finite field linear algebra subroutines*, Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation (New York), ACM, 2002, pp. 63–74 (electronic). MR 2035233
2. Tommy Färnqvist, *Number theory meets cache locality — efficient implementation of a small prime FFT for the GNU Multiple Precision Arithmetic Library*, Master’s thesis, Stockholm University, 2005, [http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2005/rapporter05/farnqvist\\_tommy\\_05091](http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2005/rapporter05/farnqvist_tommy_05091).
3. Peter L. Montgomery, *Modular multiplication without trial division*, Math. Comp. **44** (1985), no. 170, 519–521. MR 777282 (86e:11121)
4. Victor Shoup, *NTL: a library for doing number theory (Version 5.5.2)*.
5. Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, second ed., Cambridge University Press, Cambridge, 2003. MR 2001757 (2004g:68202)